Semafori privati

Realizzazione di politiche di gestione delle risorse

- Nei problemi di sincronizzazione più semplici, come quelli visti in fino ad ora:
  - la decisione circa la possibilità per un processo di proseguire nella esecuzione dipende dal valore di *un solo semaforo* ("mutex", "spazio-disp", "messaggio-disp")
  - la scelta del processo o thread da riattivare avviene tramite l'algoritmo codificato *nella primitiva signal*; tale algoritmo deve essere generale ed efficiente, pertanto normalmente è di tipo FIFO

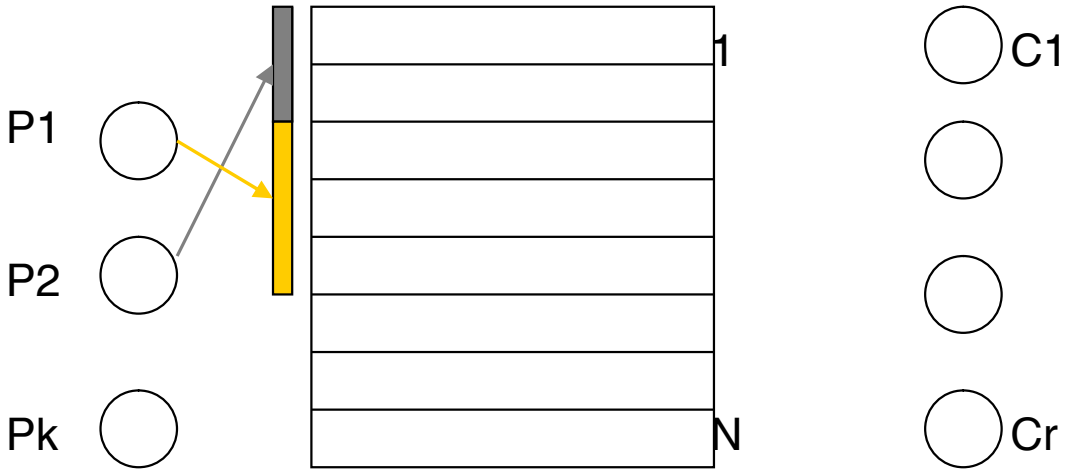
Realizzazione di politiche di gestione delle risorse

- In problemi di sincronizzazione *più complessi* è necessario realizzare politiche specifiche di acquisizione delle risorse:
  - la decisione circa la possibilità per un processo o thread di proseguire nella esecuzione dipende dal verificarsi di una *condizione di sincronizzazione* che coinvolge più variabili condivise
  - la scelta del processo da riattivare può dover avvenire sulla base di condizioni di *priorità* tra i processi
- *Politiche di gestione delle risorse devono essere programmate in modo esplicito, al di fuori delle primitive di sincronizzazione*

Esempio di politica di gestione

- Produttori-consumatori con messaggi di lunghezza variabile

- Buffer di N celle cui possono accedere più produttori e consumatori
- I produttori inseriscono messaggi di dimensione l variabile ($l \leq N$)



- Politica di gestione: Tra più produttori ha priorità di accesso quello che fornisce il messaggio di maggior dimensione

Esempio di politica di gestione

- Produttori-consumatori con messaggi di lunghezza variabile
- Per tutto il tempo in cui un produttore rimane sospeso con un messaggio di dimensione superiore allo spazio disponibile nel buffer, nessun altro produttore può depositare il proprio messaggio anche se le sue dimensioni possono essere contenute nello spazio libero del buffer
- *Condizione di sincronizzazione*:
  - "il deposito può avvenire se c'è sufficiente spazio per memorizzare e non ci sono produttori in attesa"
- Al prelievo viene riattivato tra i processi sospesi quello il cui messaggio ha la dimensione maggiore, purché esista sufficiente spazio nel buffer. Se lo spazio disponibile non è sufficiente *nessun produttore viene riattivato*

Semaforo privato

- Si definisce <u>semaforo privato</u> per il processo P_i un semaforo

  $priv_i$ (v.i. $priv_i = 0$)

- tale che:
  - $priv_i$ è inizializzato a 0
  - solo P_i può eseguire la wait($priv_i$)
  - anche altri processi possono eseguire la signal($priv_i$)

- I semafori privati sono un *pattern* utilizzato per realizzare politiche di gestione delle risorse

Realizzazione di politiche di gestione tramite semafori privati

- Acquisizione della risorsa da parte di un processo P_i:

  wait(mutex);

  <verifica della condizione di sincronizzazione

  con esecuzione condizionata di signal($priv_i$)>

  signal(mutex);

  wait($priv_i$);

  <uso della risorsa>

- P_i, se dall'analisi delle variabili comuni deduce che la propria condizione di sincronizzazione è verificata, modifica lo stato delle variabili e *si abilita* all'acquisizione della risorsa tramite una signal($priv_i$)
- Altrimenti *attende fuori dalla sezione critica*, in wait($priv_i$)

Realizzazione di politiche di gestione tramite semafori privati

- Rilascio della risorsa da parte di un processo P_j:

```
wait(mutex);
<verifica della condizione di sincronizzazione
        con esecuzione condizionata di signal(priv_i)>
signal(mutex);
```

- P_j può *scegliere*, in base alla politica richiesta, se riattivare e quale riattivare tra i processi la cui condizione di sincronizzazione è divenuta vera (grazie al rilascio della risorsa)

Realizzazione di politiche di gestione tramite semafori privati

- Proprietà della soluzione:

- La *sospensione del processo*, nel caso in cui la condizione di sincronizzazione non sia verificata, avviene *al di fuori della sezione critica* (per evitare il deadlock)
- Se è presente una competizione tra alcuni processi per l'acquisizione della risorsa, *i processi attendono su semafori distinti (privati)*
- In fase di rilascio si può quindi adottare *un qualunque algoritmo di assegnazione* della risorsa

Produttori-consumatori con messaggi di lunghezza variabile

- Soluzione con *semafori privati*
- Variabili di stato per il problema specifico:

  "sospesi": numero di processi produttori sospesi

  "vuote": numero di celle vuote del buffer

  "richiesta[i]": numero di celle richieste dal produttore Pi sospeso

- Produttori: eseguono una procedura di acquisizione e deposito nel buffer
  - *procedure* acquisizione(l: integer, i: 1..num-prod);
- Consumatori: rilasciano il buffer e realizzano la politica di risveglio dei processi sospesi o prenotati
  - *procedure* rilascio(m: integer);

Produttori-consumatori con messaggi di lunghezza variabile

- *procedure* acquisizione(l: integer, i: 1..num-prod);

```
wait(mutex);
  if  sospesi = 0 and vuote ≥ l {
            vuote := vuote - l;
            signal(priv_i);      }
  else {  sospesi := sospesi + 1;
            richiesta[i] := l;  }
signal(mutex);
wait(priv_i);
wait(mutex1);
  <deposito nel buffer>
signal(mutex1);
```

Condizione di sincronizzazione

Caratteristica del problema! Condizione in base a cui il thread può proseguire

Produttori-consumatori con messaggi di lunghezza variabile

- *procedure* rilascio(m: integer);

```
wait(mutex);
  vuote := vuote + m;
  exit := false;
  while  sospesi <> 0 and not exit do {
           <individuazione tra i processi sospesi del processo Pk
             con la massima richiesta, max>
           if  max ≤ vuote {
                    vuote := vuote - max;
                    richiesta [k] := 0;
                    sospesi := sospesi - 1;
                    signal(privk);  }
           else  exit := true;
           }
signal(mutex);
```

Mutua esclusione con risorse equivalenti e priorità tra i processi

- Politica di gestione: priorità statica tra i processi
  - {P1, P2, ... , Pn} utilizzano {R1, R2, ... , Rm}
  - Ogni processo può utilizzare una qualunque delle risorse → *risorse equivalenti*
  - A ciascun *processo* è associata una priorità

- Condizione di sincronizzazione: "Esiste una risorsa disponibile"

- In fase di riattivazione di processi sospesi viene scelto quello a cui corrisponde la *massima priorità*

Mutua esclusione con risorse equivalenti e priorità tra i processi

- Variabili di stato del problema:

"sospesi":	numero di processi richiedenti sospesi
"PS[i]":	indica se il processo Pi è sospeso
"R[j]":	indica se la risorsa j è occupata
"RA[i]":	indice della risorsa assegnata al processo Pi
"disponibile":	numero di risorse disponibili

- Procedure di acquisizione e rilascio della risorsa

Mutua esclusione con risorse equivalenti e priorità tra i processi

- Acquisizione della risorsa

```
wait(mutex);
     if  disponibile > 0 {
                    <seleziona una risorsa k tra quelle disponibili
                              utilizzando il vettore R>
                    disponibile := disponibile - 1;
                    R[k] := 1;
                    RA[i] := k;
                    signal (priv_i);  }
     else  {
                    sospesi := sospesi + 1;
                    PS[i] := 1; }  // <- oppure qui settare la priorità
signal(mutex);
wait(priv_i);
<uso della risorsa RA[i]>
```

Mutua esclusione con risorse equivalenti e priorità tra i processi

- Rilascio della risorsa

```
wait(mutex);
  if  sospesi <> 0 {
        <seleziona il processo Pi con priorità max tra quelli sospesi
          utilizzando il vettore PS>
        PS[i] := 0;
        RA[i] := m;
        sospesi := sospesi - 1;
        signal(priv_i);
        }
  else  {
        R[m] := 0;
        disponibile := disponibile + 1;
        }
signal(mutex);
```

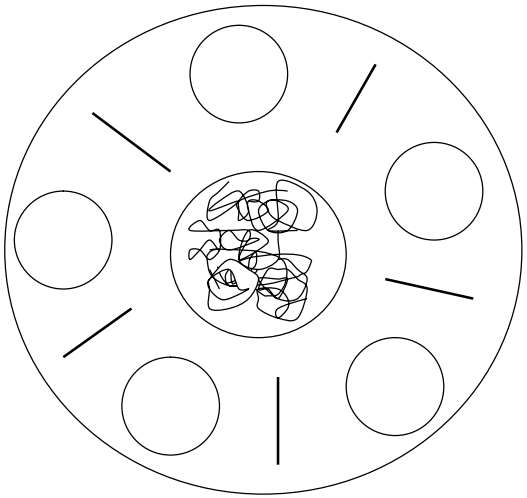
Considerazioni sull'uso dei semafori privati

- Ogni processo che durante l'acquisizione trova la risorsa non disponibile *deve lasciare traccia in modo esplicito della propria sospensione* entro la sezione critica
  - incremento di "sospesi" in entrambi gli esempi e modifica della variabile che descrive lo stato del processo, "richiesta[i]" e "PS[i]" rispettivamente
- La fase di assegnazione della risorsa è separata dalla fase di uso. Occorre quindi *registrare in modo esplicito* entro la sezione critica *la non disponibilità della risorsa* prenotata
  - nel primo esempio viene decrementata la variabile "vuote", nel secondo viene modificata la variabile R
- Il processo che libera la risorsa deve *eseguire signal($priv_i$) solo se esistono processi sospesi* che soddisfano le condizioni per il risveglio
  - variabile "sospesi" in entrambi gli esempi

Problemi classici di sincronizzazione

Filosofi a cena

- N filosofi dal robusto appetito, che alternano pensiero e cibo; una grande teglia di spaghetti al centro
- N chopstick: per cibarsi i filosofi devono utilizzare due chopstick (?? – cucina olandese)
- I chopstick devono essere utilizzati in modo esclusivo



Filosofi a cena

```
var chopstick: array [0 .. N - 1] of semaphore  initial 1;
  Philosopher_i:
    repeat
          < think >
          wait(chopstick[i]);          /* acquisizione chopstick */
          wait(chopstick[(i+1) mod N]);
          < eat >
          signal(chopstick[i]);     /* rilascio chopstick */
          signal(chopstick[(i+1) mod N]);
forever
```

Filosofi a cena

```
var chopstick: array [0 .. N - 1] of semaphore  initial 1;
  Philosopher_i:
    repeat
          < think >
          wait(chopstick[i]);          /* acquisizione chopstick */
          wait(chopstick[(i+1) mod N]);
          < eat >
          signal(chopstick[i]);     /* rilascio chopstick */
          signal(chopstick[(i+1) mod N]);
forever
```

- Inconvenienti?
- Quali soluzioni alternative basate su semafori?

- Problema classico, molte soluzioni proposte!

Filosofi a cena: soluzione con semafori privati

- *var* state: *array* [0 .. N - 1] *of* (thinking, hungry, eating)
  initial thinking;
  mutex: semaphore *initial* 1;
  priv: *array* [0 .. N - 1] *of* semaphore *initial* 0;

Filosofi a cena: soluzione con semafori privati

- Philosopher_i: // dp.pickup: protocollo acquisizione chopstick

```
wait(mutex);
state[i] := hungry;
if (state[(i+N-1) mod N] <> eating) && (state[(i+1) mod N] <> eating) {
        state[i] := eating;
        signal(priv[i]);
}
signal(mutex);
wait(priv[i]);
```

Filosofi a cena: soluzione con semafori privati

- Philosopher_i: // dp.putdown: protocollo rilascio chopstick

```
wait(mutex);
state[i] = thinking;
if (state[(i+N-1) mod N]==hungry) && (state[(i+N-2) mod N]<>eating) {
        state[(i+N-1) mod N] = eating;
        signal(priv[(i+N-1) mod N]);
}
if (state[(i+1) mod N]==hungry) && (state[(i+2) mod N]<>eating) {
        state[(i+1) mod N] = eating;
        signal(priv[(i+1) mod N]);
}
signal(mutex);
```

Filosofi a cena: soluzione con semafori privati

- La soluzione evita il deadlock, non la *starvation:*
  - se i filosofi *i-1* e *i+1* si alternano a tavola, il filosofo *i* rischia di restare sempre hungry
- Altre possibili soluzioni:
  - protocolli di acquisizione diversi per filosofi pari e dispari
  - limitare il numero massimo di commensali a N-1
- Non tutte le soluzioni danno luogo alla medesima *efficienza*
  - limitare il numero di commensali ad N-1 può determinare, nel caso peggiore, l'accesso di un solo filosofo al tavolo per il pasto
  - la soluzione mediante semafori privati consente comunque l'accesso concorrente di ≈ N/2 filosofi
  - la soluzione mediante semafori privati evita il deadlock basandosi su una tecnica generale denominata *prevenzione statica*